Exercise_1_solved

January 20, 2025

1 The (extended) pairing Hamiltonian

[1]:
```
import numpy as np
import scipy
import matplotlib.pyplot as plt
```

The pairing Hamiltonian is a schematic, but powerful Hamiltonian used to explore the emergence of superfluidity. It is a standard benchmark problem for quantum many-body approaches as it has been exactly solved by Richardson [Richardson, Phys. Lett. 3, 227 (1963)] and poses significant challenges for many methods due to a phase transition to a superfluid phase for sufficiently attractive pairing interaction. It is also simple enough to be an instructive way to learn and explore quantum many-body methods.

The pairing Hamiltonian describes spin-1/2 fermions occupying discrete evenly-space levels. The Hamiltonian to describe this is:

$$H_1 = \Delta\epsilon \sum_p \sum_\sigma (p-1) a^\dagger_{p\sigma} a_{p\sigma}.$$

$\Delta\epsilon > 0$ is the spacing between levels. p is the quantum number for the levels from $p = 1$, ..., $p = p_{\mathrm{max}}$. σ indicates whether a fermion is spin up ($\sigma = +1$) or spin down ($\sigma = -1$).

1.0.1 Problem 1: Single-particle basis

Write a function to build a list of all possible states $|p\sigma\rangle$ a single fermion can be for a given p_{max}.

[2]:
```
def make_list_of_single_particle_states(pmax):
    '''
    Creates a basis of single-particle states for the pairing Hamiltonian given␣
↪a number of levels pmax.

    Args:
        pmax: integer >= 1 indicating the number of levels

    Returns:
        List of states (p, sigma) in the single-particle basis.
    '''

    states = []
    # My solution
    for p in range(1, pmax + 1):
        for sigma in [-1, 1]:
            states.append((p, sigma))

    return states
```

[3]:
```
# This is code to test your implementation. It is not comprehensive.
for pmax in [2, 4, 8]:
  assert len(make_list_of_single_particle_states(pmax)) == pmax * 2
```

This is your single-particle basis. All remaining operators and states can be constructed from this single-particle basis.

> Note: The order of states in your basis is arbitrary. The result of your calculations will not matter how you order your basis *as long as you do everything consistently.* However, you can make life easier for yourself by choosing a well motivated ordering (for example, sorting states from lowest to highest energy).

1.0.2 Problem 2: 1-body Hamiltonian

Write a function to construct the matrix elements of the 1-body Hamiltonian above, H_1:

$$H_{ij} = \langle i = (p\sigma)|H_1|j = (p'\sigma')\rangle.$$

We use the useful abbreviated notation $i = (p\sigma)$ so that we can talk about a generic state $|i\rangle$ without worrying about its quantum numbers.

[4]:
```
def make_1body_hamiltonian(basis, delta_eps):
    '''
    Builds 1-body Hamiltonian for pairing Hamiltonian given a basis and an␣
  ↪input coupling delta_eps.

    Args:
        basis: List of single-particle states (p, sigma).
        delta_eps: The one-body coupling for the pairing Hamiltonian.

    Returns:
        1-body Hamiltonian as matrix.
    '''
    dim = len(basis)
    h1 = np.zeros((dim, dim))

    # My solution
    for i in range(dim):
        for j in range(dim):
            # (i == j) -> (p_i == p_j) and (sigma_i == sigma_j)
            if i == j:
                p, _ = basis[i]
                h1[i,j] = delta_eps * (p - 1)

    return h1
```

[5]:
```
# This is code to test your implementation. It is not comprehensive.
def norm(mat):
    return np.sum(np.pow(np.abs(mat), 2))

for pmax in [2, 4, 8]:
    basis = make_list_of_single_particle_states(pmax)
    for delta_eps in [1.0, 5.0]:
        h1 = make_1body_hamiltonian(basis, delta_eps)
        assert h1.shape == (len(basis), len(basis))
        assert norm(h1 - np.diag(np.diag(h1))) < 1e-12
        assert norm(h1) > 1e-12
```

It is generally useful to check that certain symmetry properties of your Hamiltonian are fulfilled. In this case, we can check Hermiticity.

Write a function to check that your 1-body operator is Hermitian.

[6]:
```
def check_1body_operator_is_hermitian(h1):
    '''
    Checks that a 1-body operator is Hermitian based on its matrix elements.

    Args:
        h1: The 1-body matrix elements.

    Returns:
        bool: True if Hermitian, False otherwise.
    '''

    # My solution
    dim = len(h1)
    for i in range(dim):
        for j in range(dim):
            if abs(h1[i, j] - h1[j, i]) > 1e-10:
                return False

    return True
```

[7]:
```
for pmax in [2, 4, 8]:
    basis = make_list_of_single_particle_states(pmax)
    for delta_eps in [1.0, 5.0]:
        h1 = make_1body_hamiltonian(basis, delta_eps)
        assert check_1body_operator_is_hermitian(h1)
```

1.0.3 Problem 3: Computing state energies

Within a given single-particle basis, one can easily represent a Slater determinant state

$$|\Phi\rangle = a^\dagger_{i_1} \dots a^\dagger_{i_A} |0\rangle$$

using the occupation numbers n_i for the states $|i\rangle$:

$$n_i = \begin{cases} 1 & \text{if } i \in i_1, \dots, i_A, \\ 0 & \text{otherwise.} \end{cases}$$

Write a function to generate occupation numbers from a list of occupied states $[(p_1, \sigma_1)$, ..., $(p_A, \sigma_A)]$.

```
[8]: def create_state_occupation_numbers(basis, occupied_states):
         '''
         Generates list of occupation numbers for basis based on given occupied␣
      ↪states.

         Args:
             basis: List of single-particle states (p, sigma).
             occupied_states: List of occupied single-particle states (p, sigma).

         Returns:
             occupation_numbers: List of integers, where occupation_numbers[i] = 1␣
      ↪if basis[i] is in occupied_states.
         '''
         assert len(occupied_states) <= len(basis)
         occupation_numbers = [0] * len(basis)

         # My solution
         for i, state in enumerate(basis):
             if state in occupied_states:
                 occupation_numbers[i] = 1

         return occupation_numbers
```

From the occupation numbers, it is simple to compute the energy of the state:

$$E = \sum_i n_i H_{ii}.$$

Write a function to evaluate this for your 1-body Hamiltonian.

```
[9]: def evaluate_1body_energy_from_occupation_numbers(h1, occupation_numbers):
         '''
         Evaluates energy expectation value of state.
```

```
    Args:
        h1: 1-body matrix elements of Hamiltonian.
        occupation_numbers: List of occupation numbers.

    Returns:
        energy: Energy expectation value of state.
    '''
    energy = 0.0

    # My solution
    dim = len(occupation_numbers)
    for i in range(dim):
        energy += occupation_numbers[i] * h1[i, i]

    return energy
```

At this point, experiment with your choice of occupied states. Choose, for example, $p_{\max} = 4$ and consider all possible states of 4 fermions in this basis.

1. How many possible states are there?
2. Which state(s) has/have the lowest energy? Which one(s) has/have the highest energy?

[10]:
```
# My code
pmax = 4
n_fermions = 4
delta_eps = 1

# Build basis
basis = make_list_of_single_particle_states(pmax)

# Build H1
h1 = make_1body_hamiltonian(basis, delta_eps)

# Build all possible states of n-fermions
from itertools import combinations
all_possible_occupied_combinations = list(combinations(basis, n_fermions))
print(f"There are {len(all_possible_occupied_combinations)} possible states.")
# Alternatively, you can get this analytically by evaluating 8 choose 4.

# Determine min and max energy state
e_min = 0.0
state_min = None
e_max = 0.0
state_max = None
for occed_states in all_possible_occupied_combinations:
    occs = create_state_occupation_numbers(basis, occed_states)
    e = evaluate_1body_energy_from_occupation_numbers(h1, occs)
    if state_min is None or e < e_min:
        e_min = e
        state_min = occed_states
    if state_max is None or e > e_max:
        e_max = e
        state_max = occed_states

print(f"State with minimum energy {e_min}")
print(f"Occupied sp states: {state_min}")
print(f"State with maximum energy {e_max}")
print(f"Occupied sp states: {state_max}")
```

```
There are 70 possible states.
State with minimum energy 2.0
Occupied sp states: ((1, -1), (1, 1), (2, -1), (2, 1))
State with maximum energy 10.0
Occupied sp states: ((3, -1), (3, 1), (4, -1), (4, 1))
```

My solution

1. There are $\binom{8}{4} = 70$ possible states.
2. The minumum-energy state has the $p = 1, 2$ levels fully filled and has energy 2.0. The maximum energy state has the $p = 3, 4$ levels fully filled and has energy 10.0.

1.0.4 Problem 4: Two-body interactions

So far our 1-body Hamiltonian has only provided us a set of discrete levels, but no actual interactions to explore interesting physics. For this reason, the lowest energy state is easy to construct and solves the problem exactly. Adding two-body interactions allows us to bring in nontrivial physics, which will modify the exact ground-state of the system.

The two-body interaction for the pairing Hamiltonian is

$$
\begin{aligned}
H_2 &= \frac{1}{4}\sum_{p\sigma}\sum_{q\sigma'}(-g)a^\dagger_{p\sigma}a^\dagger_{p\bar{\sigma}}a_{q\bar{\sigma}'}a_{q\sigma'} \\
&= \frac{1}{4}\sum_{ijkl} H_{ijkl}a^\dagger_i a^\dagger_j a_l a_k
\end{aligned}
$$

where $\bar{\sigma} = -\sigma$ (the opposite spin). This interaction is between pairs in the same level, which gives this Hamiltonian its name, the "pairing" Hamiltonian.

> Note: The normalization of the coupling g is not always the same in the liturature. If you compare to published values, be sure that you are employing the same "definition" of g.

A function that generates the two-body Hamiltonian matrix elements

$$H_{ijkl} = \langle i = (p_i\sigma_i), j = (p_j\sigma_j)|H_2|k = (p_k\sigma_k), l = (p_l\sigma_l)\rangle$$

for this Hamiltonian is provided below.

```
[11]: def make_2body_hamiltonian_pairing_interaction(basis, g):
    '''
    Builds 2-body Hamiltonian matrix elements for a given basis and coupling g.

    Args:
        basis: List of single-particle states (p, sigma).
        g: 2-body pairing coupling (g > 0 is attractive).

    Returns:
        h2: 2-body Hamiltonian matrix elements H_{ijkl} as a 4-dimensional␣
 ↪array.
    '''
    h2 = np.zeros((len(basis), len(basis), len(basis), len(basis)))

    for i, state_i in enumerate(basis):
        p_i, sigma_i = state_i
        for j, state_j in enumerate(basis):
            p_j, sigma_j = state_j

            # Can skip because of antisymmetry
            if i == j:
                continue
            for k, state_k in enumerate(basis):
                p_k, sigma_k = state_k
                for l, state_l in enumerate(basis):
                    p_l, sigma_l = state_l

                    if k == l:
                        continue

                    # Check that we have pairs in (i, j) and (k, l)
                    if (
                        p_i == p_j
                        and sigma_i != sigma_j
                        and p_k == p_l
                        and sigma_k != sigma_l
                    ):
                        # This logic is needed for antisymmetry
                        if sigma_i == sigma_k:
                            h2[i, j, k, l] = -1 * g
                        else:
                            h2[i, j, k, l] = 1 * g

    return h2
```

It is very useful to check the symmetries of the Hamiltonian, Hermiticity and antisymmetry:

$$H_{ijkl} = H_{klij},$$
$$H_{ijkl} = -H_{jikl} = -H_{ijlk} = H_{jilk}.$$

Write functions to check these properties.

```
[12]: def check_2body_operator_is_hermitian(h2):
          '''
          Checks that a 2-body operator is Hermitian.

          Args:
              h2: 2-body matrix elements as a 4-dimensional array.

          Return:
              bool, True if Hermitian, False otherwise.
          '''
          dim = len(h2)
          for i in range(dim):
              for j in range(dim):
                  for k in range(dim):
                      for l in range(dim):
                          if abs(h2[i, j, k, l] - h2[k, l, i, j]) > 1e-10:
                              return False

          return True


      def check_2body_matrix_elements_are_antisymmetric(h2):
          '''
          Checks that 2-body matrix elements are antisymmetric.

          Args:
              h2: 2-body matrix elements as a 4-dimensional array.

          Return:
              bool, True if antisymmetric, False otherwise.
          '''
          dim = len(h2)
          for i in range(dim):
              for j in range(dim):
                  for k in range(dim):
                      for l in range(dim):
                          if abs(h2[i, j, k, l] - (-1) * h2[j, i, k, l]) > 1e-10:
                              return False
                          if abs(h2[i, j, k, l] - (-1) * h2[i, j, l, k]) > 1e-10:
                              return False
                          if abs(h2[i, j, k, l] -  h2[j, i, l, k]) > 1e-10:
                              return False
```

```
    return True
```

[13]:
```
# Code to check that h2 is antisymmetric and Hermitian.
g = 2.0
h2 = make_2body_hamiltonian_pairing_interaction(basis, g)

print(f"H2 is Hermitian: {check_2body_operator_is_hermitian(h2)}")
print(f"H2 is antisymmetric:␣
↪{check_2body_matrix_elements_are_antisymmetric(h2)}")
```

```
H2 is Hermitian: True
H2 is antisymmetric: True
```

It is still relatively simple to compute the energy of the state based on its occupations when including two-body interactions:

$$E = \sum_i n_i H_{ii} + \frac{1}{2} \sum_{ij} n_i n_j H_{ijij}.$$

Write a function to evaluate this for your 1-body and 2-body Hamiltonian.

[14]:
```
def evaluate_1and2body_energy_from_occupation_numbers(h1, h2,␣
↪occupation_numbers):
    '''
    Evaluates energy expectation value of state.

    Args:
        h1: 1-body matrix elements of Hamiltonian.
        h2: 2-body matrix elements of Hamiltonian.
        occupation_numbers: List of occupation numbers.

    Returns:
        energy: Energy expectation value of state.
    '''
    energy = 0.0

    # My solution
    dim = len(occupation_numbers)
    for i in range(dim):
        energy += occupation_numbers[i] * h1[i, i]
        for j in range(dim):
            energy += 0.5 * occupation_numbers[i] * occupation_numbers[j] *␣
↪h2[i, j, i, j]

    return energy
```

[15]:
```
# Code to check your implementation here.
# Use
# - pmax = 4
# - 4 fermions in the lowest energy configuration
# - delta_eps = 1
# - g = 2
# The value of E should be -2.0. If it is not, check your normalization of g.

# My code
pmax = 4
n_fermions = 4
delta_eps = 1
g = 2

# Build basis
basis = make_list_of_single_particle_states(pmax)

# Build H1, H2
h1 = make_1body_hamiltonian(basis, delta_eps)
h2 = make_2body_hamiltonian_pairing_interaction(basis, g)

# Make occupations
occs = create_state_occupation_numbers(basis, [(1, -1), (1, 1), (2, -1), (2,␣
 ↪1)])

e = evaluate_1and2body_energy_from_occupation_numbers(h1, h2, occs)

print(f"E = {e}")
```

```
E = -2.0
```

1.0.5 Problem 5: Solving the Hartree-Fock equations

We will now work on solving the Hartree-Fock (HF) equations for this problem. The central object we are interested in is the Fock matrix:

$$F_{ij} = H_{ij} + \sum_{kl} \rho_{kl} H_{ikjl}.$$

As you can see, this requires the 1-body density matrix for the new basis $|\bar{i}\rangle = \sum_i C_{\bar{i}i}|i\rangle$. Our starting point will always be $|\bar{i}\rangle = |i\rangle$. Still, we need to construct the 1-body density matrix, which we can do according to

$$\rho_{ij} = \sum_{\bar{i}} C_{\bar{i}i} n_{\bar{i}} C^*_{\bar{i}j},$$

where $n_{\bar{i}} = n_i$.

Write a function to compute the 1-body density matrix given a set of coefficients $C_{\bar{i}i}$ and occupation numbers $n_{\bar{i}}$.

[16]:
```python
def construct_new_density(coeffs, occupation_numbers):
    '''
    Constructs 1-body density matrix.

    Args:
        coeffs: Matrix of coefficients from the starting basis to the new basis.
        occupation_numbers: List of occupation numbers for single-particle␣
↪states in the new basis.

    Returns:
        density: 1-body density matrix representing the many-body state in the␣
↪starting basis.
    '''
    dim = len(occupation_numbers)
    assert coeffs.shape == (dim, dim)
    density = np.zeros((dim, dim))

    density = coeffs.T @ np.diag(occupation_numbers) @ coeffs

    return density
```

[17]:
```python
# Code to check density matrix here.
coeffs = np.identity(len(occs))
density = construct_new_density(coeffs, occs)
print(density)
```

```
[[1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
```

Now write a function to compute the Fock matrix F based on your Hamiltonian (with 1- and 2-body parts) and a density. For now this density is just the density of the lowest-energy state you found above.

[18]:
```python
def compute_fock_matrix_from_density(h1, h2, density):
    '''
    Computes Fock matrix from a given density.

    Args:
        h1: 1-body Hamiltonian matrix elements.
        h2: 2-body Hamiltonian matrix elements.
        density: 1-body density matrix.
    Returns:
        f: Fock matrix.
    '''
    f = np.zeros_like(h1)

    f = np.einsum("ikjl,kl->ij", h2, density) + h1

    return f
```

[19]:
```
# Code to check Fock matrix here
f = compute_fock_matrix_from_density(h1, h2, density)
print(f)
```

```
[[-2.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. -2.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0. -1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  2.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  2.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  3.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  3.]]
```

As you may have noticed, the Fock matrix is diagonal for the pairing Hamiltonian. We have already solved Hartree-Fock without doing anything! This is because the pairing Hamiltonian does not contain any 2-body interactions that actually modify the "mean field" picture. All 2-body interactions involve "pairs" so changing the single-particle basis is not beneficial in terms of energy.

For the purposes of this exercise session and learning how to solve the HF equations, we add another two-body interaction between same-spin fermions in neighboring levels.

$$H_2' = \frac{1}{4}\sum_{p=1}^{p_{\max}-1}\sum_{q=1}^{p_{\max}-1}\sum_{\sigma\sigma'} g_{\text{hop}} a^\dagger_{p\sigma} a^\dagger_{p+1\sigma} a_{q+1\sigma'} a_{q\sigma'}$$

The function to compute this is provided below:

[20]:
```
def eval_2body_hamiltonian_nonpairing_interaction_matrix_element(basis, i, j,␣
 ↪k, l, g_hop):
    '''
    Evaluates single matrix element H'_{ijkl}.
    '''
    pi, sigi = basis[i]
    pj, sigj = basis[j]
    pk, sigk = basis[k]
    pl, sigl = basis[l]

    if abs(pi - pj) != 1:
        return 0.0
    if abs(pk - pl) != 1:
        return 0.0
    if sigi != sigj:
        return 0.0
    if sigk != sigl:
        return 0.0

    # Account for antisymmetry
    as_factor = 1
    if sigi != sigk:
        as_factor *= -1
    if pi > pj:
        as_factor *= -1
    if pk > pl:
        as_factor *= -1
    return as_factor * g_hop


def make_2body_hamiltonian_nonpairing_interaction(basis, g_hop):
    '''
    Builds 2-body Hamiltonian matrix elements for the added interaction for a␣
 ↪given basis and coupling g_hop.

    Args:
        basis: List of single-particle states (p, sigma).
        g_hop: 2-body nearest neighbor coupling.

    Returns:
        h2: 2-body Hamiltonian matrix elements H_{ijkl} as a 4-dimensional␣
 ↪array.
    '''
    dim = len(basis)
    h2 = np.zeros((dim, dim, dim, dim))

    for i in range(dim):
        for j in range(dim):
            for k in range(dim):
                for l in range(dim):
                    h2[i, j, k, l] =␣
 ↪eval_2body_hamiltonian_nonpairing_interaction_matrix_element(basis, i, j, k,␣
 ↪l, g_hop)

    return h2
```

[21]:
```
# Check Hermiticity and antisymmetry here.
g_hop = -0.5
h2p = make_2body_hamiltonian_nonpairing_interaction(basis, g_hop)
```

```
print(f"H2 is Hermitian: {check_2body_operator_is_hermitian(h2p)}")
print(f"H2 is antisymmetric:␣
↪{check_2body_matrix_elements_are_antisymmetric(h2p)}")

e = evaluate_1and2body_energy_from_occupation_numbers(h1, h2 + h2p, occs)

print(f"E = {e}")

h2f = h2 + h2p
```

```
H2 is Hermitian: True
H2 is antisymmetric: True
E = -3.0
```

Now we have a Hamiltonian for which the Fock matrix is not immediately and HF equations need to be solved:

$$H = H_1 + H_2 + H_2'.$$

Check that the Fock matrix is not diagonal.

[22]:
```
# Code to show that the Fock matrix for the new Hamiltonian is not diagonal.
f = compute_fock_matrix_from_density(h1, h2f, density)
print(f)
```

```
[[-2.5  0.   0.   0.   0.5  0.   0.   0. ]
 [ 0.  -2.5  0.   0.   0.   0.5  0.   0. ]
 [ 0.   0.  -1.5  0.   0.   0.   0.   0. ]
 [ 0.   0.   0.  -1.5  0.   0.   0.   0. ]
 [ 0.5  0.   0.   0.   1.5  0.   0.   0. ]
 [ 0.   0.5  0.   0.   0.   1.5  0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   3.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   3. ]]
```

Diagonalizing the Fock matrix will give you the new transformation coefficients for an improved basis.

Write a function to diagonalize your Fock matrix (Hint: Refer to `scipy.linalg.eigh` to do this).

[23]:
```
def diagonalize_fock_matrix(f):
    '''
    Diagonalizes the Fock matrix and returns the transformation coefficients␣
↪C_{ibar, i}.$

    Args:
        f: Fock matrix.

    Returns:
        coeffs: Matrix of expansion coefficients of new basis in terms of␣
↪starting basis. Specifically,
            coeffs[ibar, :] should be the full eigenvector corresponding to the␣
↪new state |ibar> = sum_i C_{ibar, i} |i>.
    '''
    coeffs = np.zeros_like(f)

    _, coeffs = scipy.linalg.eigh(f)

    # Scipy returns our coefficients in a different format (with eigenvectors␣
↪as columns, not rows).
    coeffs = coeffs.T

    return coeffs
```

The coefficients you get from this diagonalization can be used to construct a new density matrix. We now want to be sure that this basis is actually better: "better" in the sense that the energy is lower than before. For that we need to evaluate the energy expectation value using the density, not the occupation numbers:

$$E = \sum_{ij} \rho_{ij} H_{ij} + \frac{1}{2} \sum_{ijkl} \rho_{ij} \rho_{kl} H_{ikjl}.$$

Write a function to evaluate the energy using the density matrix.

[24]:
```
def evaluate_1and2body_energy_from_density(h1, h2, density):
    '''
    Evaluates energy expectation value of state.

    Args:
        h1: 1-body matrix elements of Hamiltonian.
        h2: 2-body matrix elements of Hamiltonian.
        density: Density matrix of state.

    Returns:
        energy: Energy expectation value of state.
    '''
    energy = 0.0

    rho = density

    energy += np.einsum("ij,ij", rho, h1)
    energy += 0.5 * np.einsum("ij,kl,ikjl", rho, rho, h2)

    return energy
```

Briefly show that the density from the new basis you get from diagonalizing the Fock matrix has a lower energy than the starting density.

```
[25]: # My solution
new_coeffs = diagonalize_fock_matrix(f)
new_density = construct_new_density(new_coeffs, occs)
new_e = evaluate_1and2body_energy_from_density(h1, h2f, new_density)
print(new_e)
```

```
-3.182820625326996
```

To solve the Hartree-Fock equations, we do these steps *iteratively* (n is the iteration number):

1. Construct density matrix $\rho^{(n)}$ from coefficients $C_{\bar{i}i}^{(n)}$.
2. Evaluate energy expectation value $E^{(n)}$.
3. Build Fock operator $F^{(n)}$.
4. Diagonalize Fock operator to get new coefficients $C_{\bar{i}i}^{(n+1)}$.

This iterative procedure terminates when our final basis is no longer improving in each iteration, so $E^{(n)} \approx E^{(n-1)}$.

Write a function to put all of these ingredients together to solve Hartree-Fock.

```
[26]: def solve_hartree_fock_iteratively(h1, h2, occupation_numbers, verbose=True):
    '''
    Solve the Hartree-Fock equations for a given Hamiltonian and occupation␣
  ↪numbers.

    Args:
        h1: 1-body matrix elements of Hamiltonian.
        h2: 2-body matrix elements of Hamiltonian.
        occupation_numbers: List of occupation numbers.

    Returns:
        e: HF energy.
        density: HF density matrix.
        coeffs: HF basis coefficients.
        n_iters: Number of iterations required to reach convergence.
    '''
    dim = len(occupation_numbers)
    assert h1.shape == (dim, dim)
    assert h2.shape == (dim, dim, dim, dim)

    e = 0.0
    rho = np.zeros((dim, dim))
    coeffs = np.zeros((dim, dim))
    n_iters = 0

    # My solution
    max_iters = 500
    energy_convergence_criterion = 1e-4
    occs = occupation_numbers

    coeffs = np.identity(dim)
    rho = construct_new_density(coeffs, occs)
    e_prev = evaluate_1and2body_energy_from_density(h1, h2, rho)

    for i in range(max_iters):

        f = compute_fock_matrix_from_density(h1, h2, rho)
        coeffs = diagonalize_fock_matrix(f)
        rho = construct_new_density(coeffs, occs)

        e = evaluate_1and2body_energy_from_density(h1, h2, rho)

        if abs(e - e_prev) < energy_convergence_criterion:
            break
        else:
            if verbose:
                print(f"HF iter {n_iters}: E_{n_iters} = {e_prev} -> E_{n_iters␣
 ↪+ 1} = {e}")
            n_iters += 1
            e_prev = e


    return e, rho, coeffs, n_iters
```

[27]:
```
# Code to check HF result for specific values of couplings.
e, rho, coeffs, n_iters = solve_hartree_fock_iteratively(h1, h2f, occs)

print(f"HF solved in {n_iters} iterations, E_HF = {e}")

print(coeffs)
```

```
HF iter 0: E_0 = -3.0 -> E_1 = -3.182820625326996
HF iter 1: E_1 = -3.182820625326996 -> E_2 = -3.2291600490220813
HF iter 2: E_2 = -3.2291600490220813 -> E_3 = -3.2436354990332346
HF iter 3: E_3 = -3.2436354990332346 -> E_4 = -3.248583220588197
HF iter 4: E_4 = -3.248583220588197 -> E_5 = -3.2503197284350893
HF iter 5: E_5 = -3.2503197284350893 -> E_6 = -3.2509332006422293
HF iter 6: E_6 = -3.2509332006422293 -> E_7 = -3.251150218310623
HF solved in 7 iterations, E_HF = -3.2512269984841575
[[ 0.00000000e+00 -9.70068097e-01 -2.39786103e-16 -9.07150573e-16
   0.00000000e+00  2.42833043e-01  4.63606022e-18  3.93329403e-17]
 [ 9.70068097e-01  0.00000000e+00  1.49071802e-16 -2.64238964e-17
  -2.42833043e-01  1.06144956e-17 -4.56561896e-17 -8.64487769e-18]
 [ 0.00000000e+00 -1.44705446e-15  5.79164962e-01  8.13505460e-01
   2.74293900e-16  3.67034124e-16 -3.05618409e-02 -4.29277081e-02]
 [ 0.00000000e+00 -4.64366890e-16 -8.13505460e-01  5.79164962e-01
  -5.69404518e-16  1.71132472e-16  4.29277081e-02 -3.05618409e-02]
 [ 0.00000000e+00 -2.42833043e-01  3.18308841e-17  2.54830257e-16
  -5.55111512e-17 -9.70068097e-01 -3.88395149e-17  2.34406137e-16]
 [ 2.42833043e-01  0.00000000e+00 -5.95511209e-16  1.05558035e-16
   9.70068097e-01 -4.24027282e-17  1.82387093e-16  3.45345095e-17]
 [ 0.00000000e+00  4.48236376e-17 -1.18162427e-02  5.13535846e-02
  -5.38603304e-18  2.77864202e-16 -2.23924789e-01  9.73180803e-01]
 [ 0.00000000e+00  4.24784106e-18 -5.13535846e-02 -1.18162427e-02
   1.60704609e-16 -2.93950821e-17 -9.73180803e-01 -2.23924789e-01]]
```

Congratulations, you have solved Hartree-Fock!

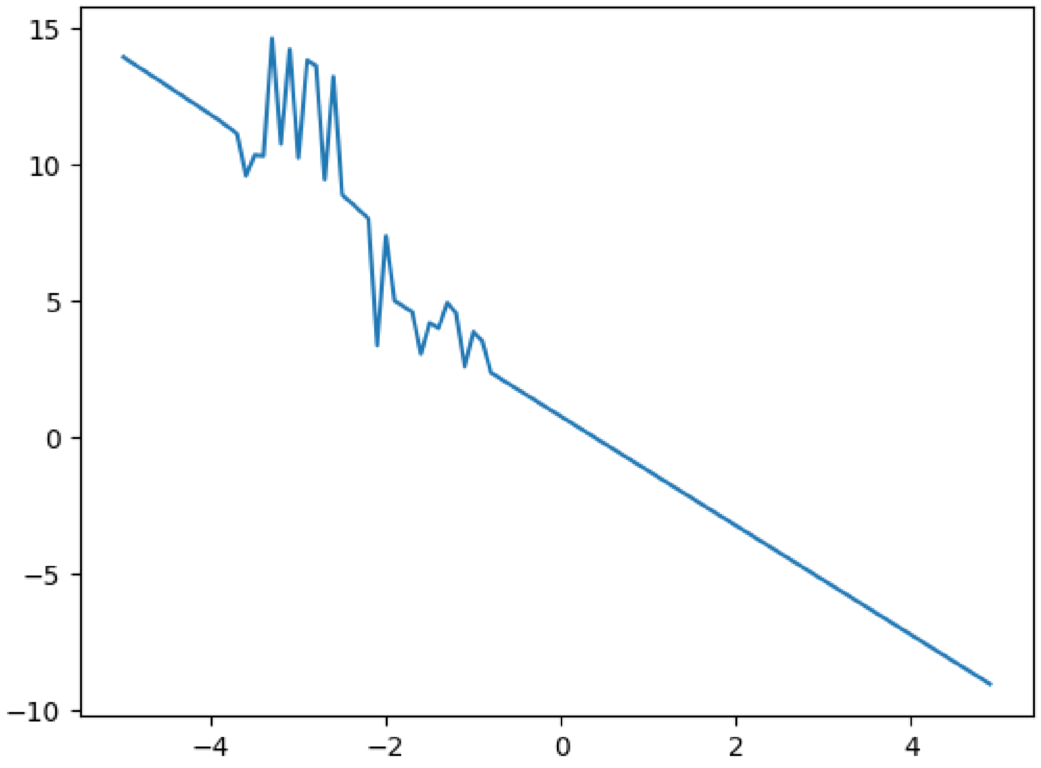
Explore how the energy depends on $g/\Delta\epsilon$ and $g_{\text{hop}}/\Delta\epsilon$. How does the number of iterations depend on g, g_{hop}? Do you have problems with convergence in any cases?

[28]:
```python
# My code
pmax = 4
n_fermions = 4
delta_eps = 1

# Build basis
basis = make_list_of_single_particle_states(pmax)

# Build H1
h1 = make_1body_hamiltonian(basis, delta_eps)

# Make occupations
occs = create_state_occupation_numbers(basis, [(1, -1), (1, 1), (2, -1), (2,␣
 ↪1)])

# Fix g_hop to -0.5
g_hop = -0.5
h2p = make_2body_hamiltonian_nonpairing_interaction(basis, g_hop)

# Vary g in somewhat reasonable range
g_vals = np.arange(-5.0, 5.0, 0.1)
e_vals_g = []
n_iters_g = []
for g in g_vals:
    h2 = make_2body_hamiltonian_pairing_interaction(basis, g)

    h2f = h2p + h2
    e, rho, coeffs, n_iters = solve_hartree_fock_iteratively(h1, h2f, occs,␣
 ↪verbose=False)

    e_vals_g.append(e)
    n_iters_g.append(n_iters)
```

```
# Fix g to 2.0
g = 2.0
h2 = make_2body_hamiltonian_pairing_interaction(basis, g)

# Vary g_hop in somewhat reasonable range
g_hop_vals = np.arange(-5.0, 5.0, 0.1)
e_vals_g_hop = []
n_iters_g_hop = []
for g_hop in g_hop_vals:
    h2p = make_2body_hamiltonian_nonpairing_interaction(basis, g_hop)

    h2f = h2p + h2
    e, rho, coeffs, n_iters = solve_hartree_fock_iteratively(h1, h2f, occs,␣
 ↪verbose=False)

    e_vals_g_hop.append(e)
    n_iters_g_hop.append(n_iters)
```

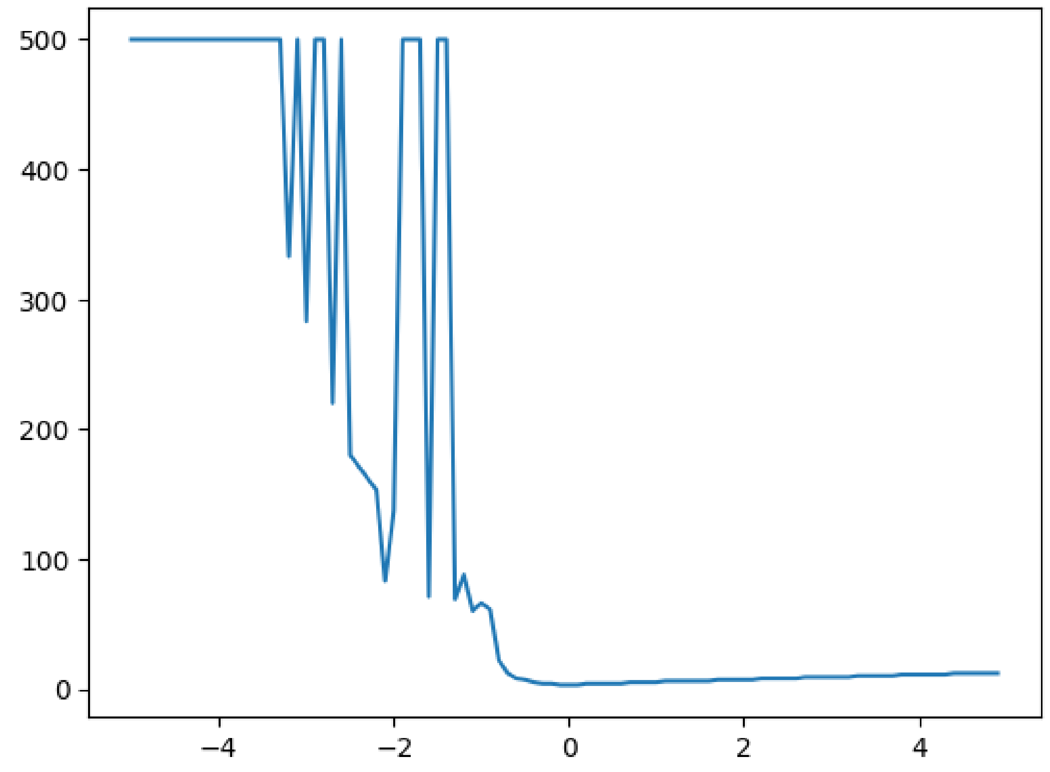
```
[29]: plt.plot(g_vals, e_vals_g)
```

```
[29]: [<matplotlib.lines.Line2D at 0x1157b2750>]
```



[30]:
```
plt.plot(g_vals, n_iters_g)
```

[30]:
```
[<matplotlib.lines.Line2D at 0x1170d4740>]
```



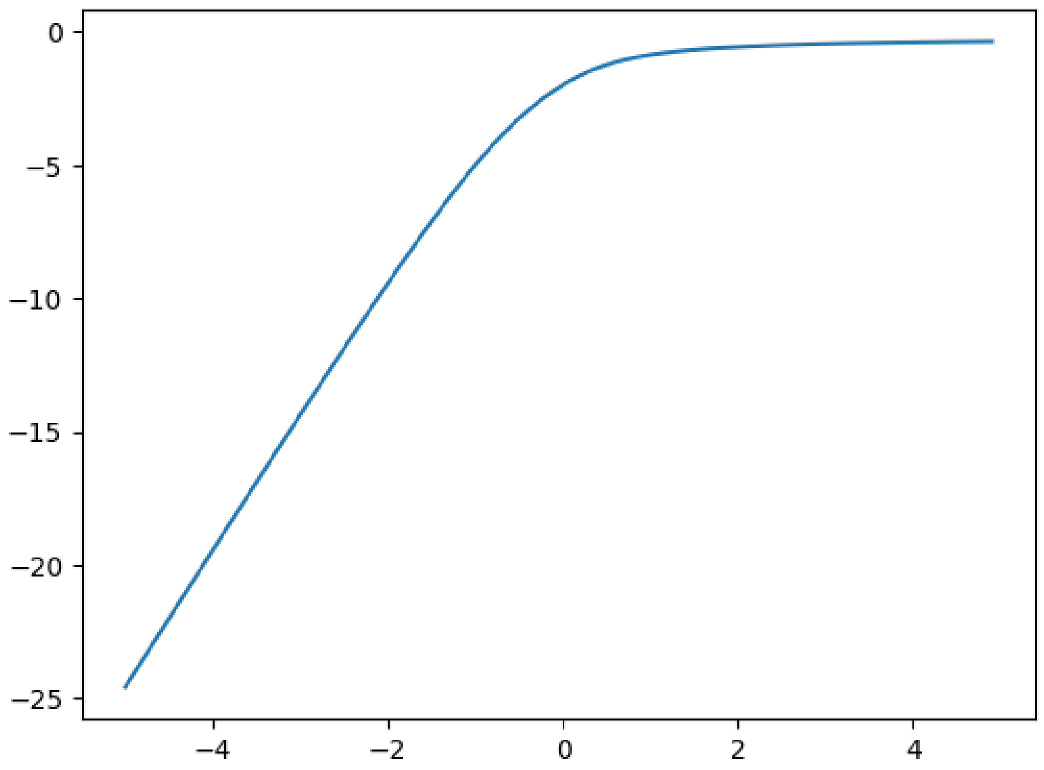
We see that for repulsive values of g $(g < 0)$ we quickly run into convergence issues.

This can be resolved by having a more sophisticated update scheme for the density matrix, for example simple mixing:

$$\rho = (1 - \alpha)\rho^{(i-1)} + \alpha\rho^{(i)}.$$

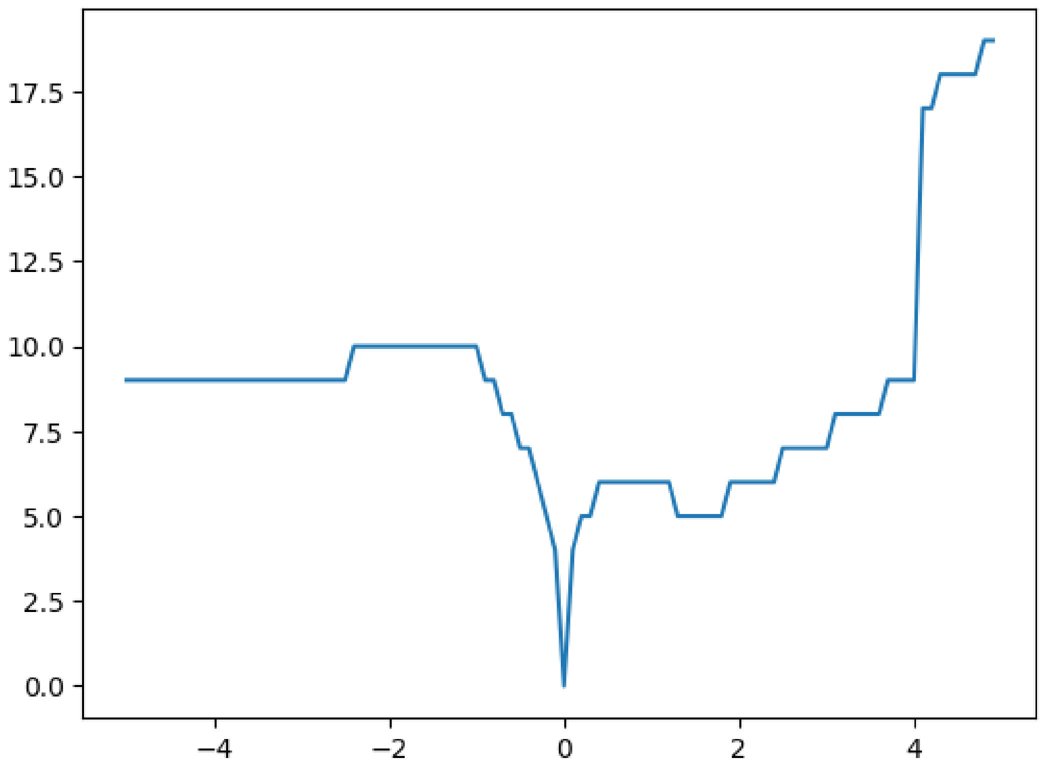
[31]:
```
plt.plot(g_hop_vals, e_vals_g_hop)
```

[31]:
```
[<matplotlib.lines.Line2D at 0x11710eb10>]
```



```
[32]: plt.plot(g_hop_vals, n_iters_g_hop)
```

```
[32]: [<matplotlib.lines.Line2D at 0x117182a80>]
```


17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−4
−2
0
2
4

```
[ ]:
```